AFRL-RI-RS-TR-2011-123

LIVE INFORMATION OBJECTS

CORNELL UNIVERSITY

JUNE 2011

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2011-123 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
ALBERT FRANTZ Work Unit Manager	JULIE BRICHACEK, Chief Information Systems Division Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
June 2011	Final Technical Report	October 2008 – December 2010

4. TITLE AND SUBTITLE

LIVE INFORMATION OBJECTS

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-09-1-0003

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Kenneth P. Birman

5d. PROJECT NUMBER
LIFO

5e. TASK NUMBER
09

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Cornell University
Upson Hall 4119B
Ithaca NY 14853

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RISE
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2011-123

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2011-1983
Date Cleared: 5 APRIL 2011.

13. SUPPLEMENTARY NOTES

14. ABSTRACT: The Cornell Live Information Objects (LIO) effort was undertaken to overcome a limitation of the tools used to create modern information-enabled applications. When using the Global Information Grid (GIG) standards, existing technologies assume continuous connectivity to some form of data center. As a result, while it is not difficult to build powerful information-enabled digital dashboard applications that are sharable, those applications become unavailable if a user lacks a high-speed connection to the base system. For forward-deployed units in challenged networking environments users may have connectivity via tactical network links, but reach-back to the base is often poor or non-existent. Our project eliminated the reach-back dependency and created a new distributed collaboration architecture that supports LIO. It is easily extended, and permits even a non-programmer to create and share live collaboration tools. Users see one-another's updates in a secure, synchronized, fault-tolerant, and extremely fast manner. Fundamental research contributions were made at the level of the language, the system, the type-checking scheme used, the multicast protocols required for fast updates, and the underlying model. The technology is available under FreeBSD licensing.

15. SUBJECT TERMS
Distributed Programming, Distributed Computing, Information Mashups, Distributed Service Oriented Collaboration, Digital Dashboard, Command and Control, Scalable Secure Real-time Multicast, .NET, Typed Objects

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			ALBERT FRANTZ
U	U	U	UU	33	19b. TELEPHONE NUMBER (*Include area code*) N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

Section **Page**

List of Figures

1.0 SUMMARY AND INTRODUCTION

1.1 Summary - Context of this Effort

The Cornell Live Information Objects ("live objects" for short) effort was undertaken to overcome a limitation of the tools used to create modern information-enabled applications.

The US military has standardized on what are called the Global Information Grid (GIG) architectural standards, which are a hardened version of the same Web Services (WS) standards used in today's web applications, such as search, e-Commerce, "cloud computing", etc. These technologies support sharing of information and collaboration but only for users who have connectivity to some form of data center. As a result, while it is not difficult to build powerful information-enabled digital dashboard applications that are sharable, those applications become unavailable if a user lacks a high-speed connection to the base system. For forward-deployed units, aircraft, or other kinds of uses in challenged networking environments, users may have connectivity to one-another via tactical network links, but reach-back to the base is often poor or non-existent.

The Cornell live objects work eliminates the reach-back dependency by supporting a powerful new way of architecting distributed collaboration systems. This *distributed collaboration architecture* [9][12][14][15] is easily extended, and permits even a non-programmer to create and share secure, responsive, live collaboration tools that can adapt to run under widely varied conditions. When users share the same object and capture information, for example images, all see one-another's updates in a secure, synchronized, fault-tolerant, and extremely fast manner. Data travels directly from user to user, rather than indirectly through a data center.

While the live objects distributed collaboration architecture is an important contribution of this work, as is the prototype platform, our effort went much further. Fundamental research contributions were made at the level of the language, the system, the type checking scheme used, the multicast protocols required for fast updates, and the underlying model. The technology is available for public use under FreeBSD (Free Berkeley Standard Distribution) public domain licensing.

1.2 Introduction - Core Research Objectives

Our effort focused on the following list of core research objectives, all of which were accomplished over the course of the two year research program. We proposed to develop a new form of live information management framework for coordinated Integrated Communications and Control (IC2) planning.

We indicated that the core capabilities of the platform would include:

1. A component-based object-oriented architecture and platform for constructing complex applications by composing smaller "live objects" that encapsulate interactive GUI (Graphical User Interface) functionality, event-processing logic, protocols (including replication protocols) and media feeds.
2. Consistent use of drag-and-drop interfaces throughout the system, to facilitate creation of new decision-making support tools and adaptation or extension of existing tools, often by non-programmers working in the field.
3. A powerful "properties language" that can endow live objects with sophisticated reliability, coordination and synchronization protocols, expressed in a new high-level language. Not only can this language express important existing concepts, but it also facilitates implementing new ones down the road.
4. Synchronization mechanisms to assist in coordination when complex human-in-the-loop decision processes are mediated by applications built using live objects.
5. Hierarchical tools for building new objects by mutating existing information objects, for example by replacing object members with other compatible members, modifying embedded parameters, or replacing existing event handling logic with new logic.
6. Compatibility with SOA (Service Oriented Architecture) technologies such as Web Services and ESB (Enterprise Service Bus).
7. Cross-platform compatibility with Windows, Linux and other systems.

In agreeing to fund the work, AFRL (Air Force Research Laboratory) stipulated that as a part of the activity, we enter into close dialog with "owners" of C2 (Command and Control) digital dashboard applications and architectures. We did so, exploring a number of standard military solutions and also creating close working relationships with colleagues at AFRL and also in the office of the Air Force Chief Information Officer (CIO) and Chief Technology Officer (CTO). We briefed two different CIOs (Mr. Gilligan and Mr. Tillotson) and one CTO (Mr. Werner), and they, in turn, briefed the current CIO (General Lord) and various teams that they designated through a primary technical liaison, Dr. Coimbitore (Sekar) Chandersekaran. Through these contacts and others, we identified specific scenarios of strong interest to the Air Force and were able to turn those into less sensitive challenge problems suitable for study in an academic setting with student involvement, mocked up the solutions, and demonstrated them [1][2]. As this effort ends, the current CIO (General Lord) is exploring options for transitioning the technology to a vendor who would be tasked with incorporating the prototype into a JFEX (Joint Forces Expeditionary Exercise) exercise late in 2011 or early in 2012. Cornell would subcontract to assist in this activity, if it occurs.

As noted, our solutions are real and available for download under a FreeBSD license with no patent or other restrictions. Our publications document the hard problems we solved and represent a roadmap that others could follow to extend their own digital dashboards with the same kinds of GIG-compatible distributed collaboration capabilities seen in our solutions.

2.0 METHODS, ASSUMPTIONS, AND PROCEDURES - OVERVIEW AND MAJOR FINDINGS

Developers who work with modern component integration and productivity tools can create desktop applications faster and more easily than ever before. These tools promote a style of development in which the language, runtime environment, debugger and profiler create a seamless whole. The military has ambitious plans to use information dashboards in many settings, including for purposes such as battlefield coordination, search and rescue, and in support of forward deployed units. The vision is compelling and has wide applicability at many levels of military planning and task execution. Broadly, planners create information applications using a "drag and drop" style, then distribute them to users, who can carry the application into the field and collaborate by interacting with the applications from devices with all sorts of form-factors and interfaces.

This compelling vision runs up against limitations of the GIG standards [12][11][14][15], widely adopted by the industry and supported by enhanced versions of the same powerful development tools used to create web applications for commercial networking purposes. The problem is that because the commercial network focuses on continuous connectivity, the tools for building collaboration tools tend to depend upon the data center itself, which plays various critical roles. For example, when updates occur, the typical GIG system is expected to relay them into the data center, which then uses a form of data distribution system called an enterprise service bus (ESB) to distribute the data to servers connected to other users. The data is then relayed back out. Thus, lacking connectivity, the application freezes, even if the collaborating users are physically close to one another and have good networking links to each other. Moreover, even with perfect connectivity, those links back to the data center may be long and slow, and scalability of ESBs is known to be very poor [18]. Thus, with more and more users generating updates, these traditional approaches tend to slow down. They work well only if there is only a single source for each kind of update, and that pattern, while common and important, is just one of many patterns seen in the kinds of important scenarios we analyzed.

This means that if an application would be used by disconnected users or by forward-deployed units with poor connectivity back to the data center (and here users might include aircraft,

troops on the ground or on vehicles, weapons with integrated targeting or other technology), the application will be slow, will often freeze up, and may have very limited functionality. In the standard GIG architecture, a disconnected application is useless.

Consider a typical information-enabled digital dashboard application [1]. A forward deployed unit has been tasked to search a village for insurgent activity. Troops have maps, and spread out to conduct the search. As they work, they capture images, mark up the map with notations ("clear", "possible bobby trap", "insurgent stronghold", "elementary school", etc). Ideally, this data would be shared, forming a unified, replicated, application "state" that is automatically and consistently updated when changes are made, and that enforces security policies as needed and appropriate. Perhaps they come under fire; crouched behind walls and in doorways, the soldiers consult their applications to see where the snipers are located and the tool should help them divide up the task of responding. It might integrate ground and air assets: the Army unit on the ground could share information with an F-16 squadron overhead, which flies in to take out an insurgent stronghold while avoiding the locations of our troops and the Iraqi soldiers working with them. Information tools, if they can be customized and if they work properly, can transform the battlefield.

Constraints such as the GIG reach-back connectivity requirement cripple this application. With the connectivity rule, the moment our team leaves their home base, they lose network reach-back to the data center and hence the application becomes frozen in whatever state it had at that instant. No sharing can occur unless the troops happen to have extremely fast tactical network links back to the base: a completely unrealistic requirement that is rarely (if ever) satisfied in the field. Thus, the GIG development tools deny our military a critical and extremely powerful form of information-enabled tool for the planner, the warfighter, and for coordination with our allies.

Yet it turns out that this reach-back requirement isn't fundamental. The GIG standards can interoperate with "edge" protocols; commercial vendors such as Google simply haven't found it worthwhile to offer that possibility to their application developers [12][14]. Our project at Cornell undertook to fill the gap, and in the process we created a new information architecture that builds on the GIG standards but focuses on distributed collaboration. Doing so required us to research the underlying issues (which pose rich scientific questions), and to apply our prototype system to real problems obtained through dialog and collaboration with colleagues at AFRL and in the office of the Air Force Chief Information Officer and Chief Technology Officer, who both became keenly interested in the work we did and are actively exploring transitioning opportunities into major military systems. Our Cornell effort has ended as of December 2010, but we believe that the Live Objects platform has already had a real impact on the Air Force and on the thinking of the research community, as well.

Central to our work has been the recognition that only by defining an architecture capable of integrating direct sharing between collaborating users of a live objects application with GIG services running in data centers would it be possible to satisfy all our goals. The insight comes down to this: a great deal of data already exists and lives within GIG-compliant data centers. But much of the information collected as an application is used in the field by forward deployed units will be exchanged over tactical links in a peer-to-peer manner. Thus, we simply need to find ways for these two styles of communication to coexist.

For example, consider the scenario illustrated in Figure 1, which is intended to evoke the movie Minority Report, in which a police unit's members worked together to identify the location of a crime so as to intervene and prevent it. Users of the system pulled information of various kinds into the shared scenario: city maps, images, data from databases, videos from real-time feeds in the city itself. Working together, they rapidly solved their problem. The live application could then be carried into the field as the unit deployed to prevent the crime. In our earlier scenario, we might combine maps and other intelligence, previously prepared by intelligence analysts, with data collected in real-time as the soldiers sweep the village, clear buildings, come under fire, defeat insurgents, etc. This mixture of data center and peer-to-peer information is inevitable. By ignoring half the story, existing GIG-based platforms stand on one leg.

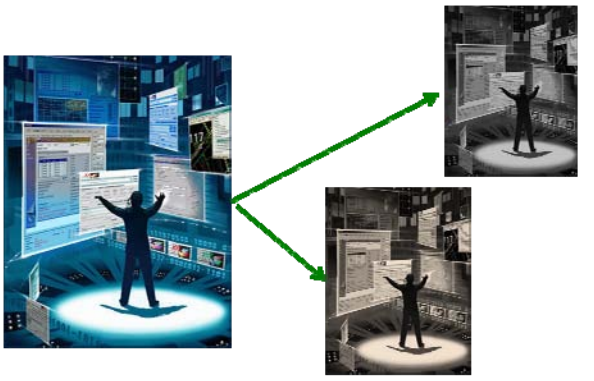

Figure 1 Sharing a Live Object.

Cornell's live object platform enables the creation of objects that are best understood as distributed mechanisms whereby a group of software components can communicate with each other, share data, exchange events, or coordinate actions. Unlike the usual GIG platforms, these objects can communicate *in any way that makes sense for the developer.* Often, this sharing occurs in a decentralized, peer-to-peer fashion, but it can also use the standard GIG approach of fetching data from a data center or relaying data through it. Thus, Cornell's platform is a bit like the one used in Minority Report, and very different from the platforms currently available from the vendors providing GIG solutions. Yet, as we'll see further below, the solution is fully GIG compliant. We simply go beyond the point at which existing GIG platforms stop: we took an important next step, solving the research questions it posed.

A live object can represent, for example, a streaming video, a news channel, the weather, a map, intelligence information about a set of targets, data about troop locations, data about hazards, collaborative documents, replicated variables, or even fault-tolerant services. The

approach is intended to scale. The platform was conceived with the goal of showing that it is possible to create completely new forms of collaboration and coordination applications for a wide range of uses and for a wide range of deployment environments, including autonomous, self-managed applications that could literally span the globe, all the time respecting GIG standards even as we go well beyond what they are traditionally used to accomplish today. Obviously, the applications of interest demand high security and other assurance guarantees; our solution responds using a novel form of type checking in which type assertions and type exceptions play the roles of access control and the blocking of unauthorized actions.

From the programmer's perspective, live objects are an encompassing solution that take a wide range of earlier technologies such as multicast, group communication, publish-subscribe, or state machine replication, and then puts them under a single umbrella: a common architecture that models all of these seemingly different and often proprietary solutions in a standardized manner, allowing them to interoperate flexibly and securely by means of strongly typed event passing. Development and debugging tools work in a natural way.

The system is easy to use. The initial creation of a new kind of live object is done by a programmer. For example, suppose that we want to create an object representing a new kind of aircraft. The programmer implements the object itself by writing code using a simple, event-driven style in C#, Java, C++, Visual Basic or some other popular language. Often the desired object will consist of two or three sub-objects that compose into a solution, with each specialized for a different role. For example, an aircraft object might have a component to track the aircraft position (by capturing coordinates from a radar, or via telemetry from the aircraft itself), a component to render the "skin" of the plane, and a component that takes the coordinates and uses them to tell the graphics engine of the display system to reorient, resize and then redisplay the aircraft.

What does "composition" mean in an object-oriented application of this sort? The basic idea is that live objects use a typed, extensible event representation to exchange data. A GPS (Global Positioning System) coordinate tracking object, for example, might have an interface through which it emits coordinate objects, one object each time it senses a significant change in the orientation of the aircraft. It "finds" the aircraft through configuration parameters supplied to the object when it is created, and those parameters, in turn, can come from the runtime environment, or be supplied by other objects. For example, in our case they would specify the GPS data source to read and the aircraft to track, if that source can track many of them.

When two objects are composed together a new object results. Composition involves taking some of the output ports of one object and binding them to input ports of other objects. Type checking is done to make sure the types match and also to perform security validation, as explained later in more detail [9][12][19]. Thus, remaining with our example, we could

compose a GPS tracker object with an airplane render object: each new coordinate would be read by the GPS tracker, packed as an event, emitted on the output port, read into the input port of the renderer object, unpacked, and then it would reoriented the aircraft. In live objects, there are standard display objects for 2D and 3D rendering: these take events (like any object) but interpret them as instructions for the underlying display system, which in our prototype uses Windows XNA (the Windows 3D graphical rendering subsystem).

Some objects are coded from scratch, but very often one can avoid doing so by taking some existing object and customizing it: changing the way it captures state information, the way it displays the object, the rendered "skin", etc. Objects can be edited, using a simple visual editor of our own design, and can be composed with one-another in a modular, flexible, manner provided that their output and input types match on the ports that are designated for the composition. In this way, a graph of interacting objects is created, representing some entity of interest. Through such steps, one creates objects representing every kind of entity seen in the deployment environment, with each object "instance" parameterized to match the appropriate object category with a means of capturing the kinds of data needed for some specific use. Other objects could represent orders, locations of friendly or enemy forces, buildings, etc. Objects can also hold groups of other objects: a city object might contain building objects; a helicopter squadron multiple helicopters, and so forth.

What about data that would be found in an information management system like a file or a database or a web service? Here, one uses live objects that employ the GIG (Web Services) standards to pull the desired information from a data center in the normal model, which (as noted repeatedly above) does entail continuous connectivity between the object and the data center; if the connection is lost, the object will experience timeouts on its requests (and should handle them in an appropriate manner that won't freeze up other objects that aren't having this problem). Our prebuilt library of live objects includes objects that fetch weather data, maps, information about buildings and points of interest, etc, from a diverse set of web sources such as Google, Microsoft, the FAA (Federal Aviation Administration), etc. Since the objects can coexist, this means that we can easily create compositions ("mashups") that combine data from Microsoft, Facebook and Google in a single context [1]. What makes these objects live is that they poll at some frequency, reissuing requests (perhaps, with revised GPS coordinates if the underlying data is in a map), and then send update events to surrounding objects.

The platform is smart about event passing and we should note that zero copying occurs when objects exchange events in this manner. Thus even a very large object like a map or a high-resolution image can be passed around very easily.

The reader should now have an image of a live object as a graph of subobjects that are connected to one-another by binding output ports to input ports and interact via event passing.

The type of an object is the set of types associated with its ports. A composed object also exposes input and output event ports with types. Once created, objects can be saved like files, or stored in databases. They have an XML (Extensible Markup Language) representation, like a web page, in which the .NET byte code source for the object can be included, but can also be identified via URL (Universal Resource Locator). This allows a new object to be downloaded, byte-code checked for safety against local platform rules (which could disable this feature), and then executed. Thus a previously unfamiliar kind of object can still be passed to a user and executed safely.

But this discussion has said nothing at all about replication. Live objects treat replication as a fundamental feature of the environment: every object is "understood" to run as a set of replicas, and live objects makes it easy for those replicas to find one-another and form communication connections via the standard IP (Internet Protocol) protocols: TCP (Transmission Control Protocol), UDP (User Diagram Protocol) and IP multicast. Live Objects doesn't require that a set of replicas connect to one-another, but the assumption is that most objects will "somehow" arrange to keep their replicas in synchronized, consistent states. For many objects this is done by composition with some prebuilt "multicast" object that takes events and then broadcasts them to the other replicas of the same object type: a broadcast within a set, hence the term "multicast".

So, returning to our earlier example of an aircraft object, rather than have the GPS capture data and hand it directly to the local rendering object (which would give functionality on a single machine but not keep replicas in sync) a more standard approach would be this. First, the GPS object would be built from what we call the "leader pattern": a type of object in which there is a set of passive replicas and one designated leader, selected by the live objects system, which plays a special role. If the leader fails or leaves the system, a new leader is picked. This ensures that just one GPS tracker is active at a time even in a group that might include 20 replicas of the GPS tracking function (one each on 20 end-user machines).

This leader reads GPS updates, as before, but now passes the events to the input port of a multicast object, which multicasts the data to its other replicas, presumably on those same 20 machines. Incoming data is passed back into the GPS object replicas (the full set of 20 passive objects plus the 1 leader), and only *now* does the composed object pair emit an event to the render. By selecting a multicast object that offers a totally ordered, secure, fault-tolerant protocol (also called an "atomic broadcast" primitive), all replicas see the same events, in the same order, and thus the aircraft exhibits identical behavior on all machines. A special "state transfer" function is used when a replica joins a preexisting group (e.g. a 21st member of the squadron launches his live objects application): this enables the object to catch up by reading state data from the leader over a special, out-of-band, TCP connection.

How does a multicast object "work"? We have several versions, developed over many years. One kind of multicast object just relays data via a standard GIG publish-subscribe message bus, in the manner of a normal, internet-hosted, web application. It performs poorly, however, because all data must relay through the data center, and freezes if the datacenter connection is lost. A second kind of multicast object uses IP multicast to send data directly from peer to peer: it offers stunning speed, but obviously can only be used when IP multicast is available. In fact it comes in two flavors: one with strong realtime guarantees (the Ricochet protocol [20]) and a second with strong logical consistency properties (the Quicksilver Scalable Multicast protocol [18]). Yet additional solutions can be devised using purely TCP, or other options. Indeed, one group of students used Twitter to implement multicast as a classroom project. That object was slow… but it worked.

Notice that even a multicast object can have various "types" of guarantees: total ordering, reliability, timing guarantees, etc. We view these as a form of types and indeed our event-based composition system can do type checking at this level of detail [13][10]. Thus, an application that requires a totally ordered and lossless multicast would throw an exception if composed with an object that only implements FIFO (First In First Out) ordering, or that has realtime guarantees but only offers probabilistic loss recovery if a packet is lost. This exception occurs at compile time if the objects are statically composed by the designer, but can also happen at runtime, if an object is designed to select from a set of candidate objects on the basis of runtime conditions.

Thus, the user effectively creates a kind of web page that can be saved, loaded and executed. Running the web page launches new replicas of the various objects in the assemblage: often tens or hundreds of them (we've tested with tens of thousands). The objects form themselves into groups, initialize via state transfer from group leaders, and then maintain tight synchronization, in whatever manner the designer selected.

Use of an object requires appropriate permissions: access permissions to even read the object and load it, but also runtime permissions to join the existing object group, and these permissions can be based on a hardware root of trust such as a CAC (Computer Access Card) card or a trusted computing chip on the user's computer. A form of type exception is thrown when security rules are violated, and the system refuses to perform the action. Thus, an unauthorized individual is prevented from accessing objects without permission, modifying them, launching them, adding information to a running scenario, etc. The policies can be fine-tuned to match the desired use and cannot be subverted without breaking the underlying .NET system security.

Because most objects are created well before being used, live objects can often be used without any programming experience at all. Imagine a squadron leader preparing his or her

team for an engagement. The leader receives tasking orders and sits down at a computer, browsing through virtual folders containing data appropriate to the situation, accessing them much as one accesses files containing images or videos, and deciding which ones bring value to the forward deployed unit members. Those useful objects are then combined to create the runtime scenario: The leader simply drags and drops the desired objects to create live documents -- web pages, slide shows, and so forth. He or she then shares the application with the individuals who will use it, who carry it into the field. Each object has some replicated state that will be seen in a consistent and coordinated manner by all users who have a replica of it. Moreover, if permitted, those users can further upload data, for example by capturing images or annotating existing objects. Given the connectivity required for that form of object, updates should be instantly and consistently replicated to other squadron members. If connectivity is lost, only those objects that depended on that form of connectivity will be impacted, and they would only "lock up" if designed to do so. For example, a map object might simply continue to display the old map rather than going blank or throwing error messages up to the user. This stands in sharp contrast to today's web services solutions, which are so dependent upon continuous connectivity that if connectivity is lost, they often crash. (As an example: the iPad has a map application, and it can function like any GPS device, tracking your car and giving driving instructions. But if the iPad loses its network connection, the map vanishes and an error box appears: not the most useful functionality if one is driving into the woods!)

What makes live objects so exciting is that they are a gateway to a completely new vision for the Web: they enable a new kind of information-enabled Web that, for the first time, is truly *active*. Unlike today's GIG, which is hard-wired to the Internet and cannot support true mobility and disconnected operations, live objects have no such restrictions: while an object that uses GIG standards would indeed freeze up without a connection (and we do support such objects), those using peer-to-peer communication remain capable of disconnected, agile, adaptive behavior even when disconnected from the military Internet.

3.0 RESULTS AND DISCUSSION - CORE ELEMENTS OF THE LIVE OBJECTS ARCHITECTURE

In this section we offer more detailed definitions of the key elements of the live objects architecture. Among our papers, [19] offers a comprehensive treatment of the topic.

We have been somewhat informal in our use of the term *live object.* More specifically, a live object is a running instance of a distributed multi-party (or peer-to-peer) protocol. Live objects have an object-oriented presentation: they are entities with a distinct identity, that encapsulate

internal state and may have threads of execution, and that exhibit a well-defined externally visible behavior that can be modeled as a “type”.

Figure 2 is an illustration of the basic concepts involved in the definition of a live distributed object.

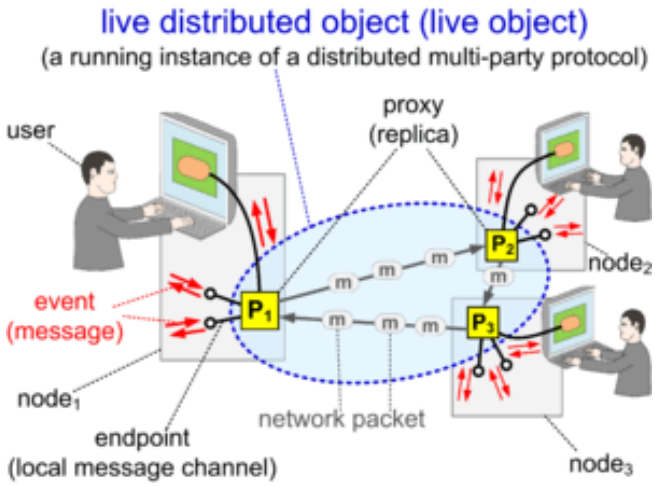

Figure 2 An illustration of the basic concepts involved in the definition of a live distributed object.

The key concepts mentioned here are defined as follows (read [10][11][12] for details):

- **Identity**. The *identity* of a live distributed object is determined by the same factors that differentiate between instances of the same distributed protocol. The object consists of a group of software components physically executing on some set of physical machines and engaged in mutual communication, each executing the distributed protocol code with the same set of essential parameters, such as the name of a multicast group, the identifier of a publish-subscribe topic, the identity of a membership service, etc. Thus, for example, publish-subscribe channels and multicast groups are examples of live distributed objects: for each channel or group, there exists a single instance of a distributed protocol running among all computers sending, forwarding, or receiving the data published in the channel or multicast within the group. In this case, the object's identity is determined by the identifier of the channel or group, qualified with the identity of the distributed system that provides, controls, and manages the given channel or group. In the case of multicast, the identity of the system might be determined, for example, by the address of the *membership service* (the entity that manages the membership of the multicast group).

- **Proxies** (**replicas**). The *proxy* or a *replica* of a live object is one of the software component instances involved in executing the live object's distributed protocol. The object can thus be alternatively defined as a group of proxies engaged in communication, jointly maintaining some distributed state, and coordinating their operations. The term *proxy* stresses the fact that a single software component does not in itself constitute an object; rather, it serves as a *gateway* through which an application can gain access to a certain functionality or behavior that spans across a set of computers. In this sense, the concept of a live distributed object *proxy* generalizes the notion of a remote procedure call stub.

- **Behavior**. The *behavior* of a live distributed object is characterized by the set of possible patterns of external interactions that its proxies can engage in with their local runtime environments. These interactions are modeled as exchanges of explicit events (messages).

- **State**. The *state* of a live distributed object is defined as the sum of all internal, local states of its proxies. By definition, it is distributed and replicated. The different replicas of the object's state may be strongly or only weakly consistent, depending on the protocol semantics: an instance of a consensus protocol will have the state of its replicas strongly consistent, whereas an instance of a leader election protocol will have a weakly consistent state. In this sense, the term *live distributed object* generalizes the concept of a *replicated object*; the latter is a specific type of live distributed object that uses a protocol such as Paxos, virtual synchrony, or state machine replication to achieve strong consistency between the internal states of its replicas. The state of a live distributed object should be understood as a dynamic notion: as a point (or *consistent cut*) in a stream of values, rather than as a particular value located in a given place at a given time. For example, the externally visible state of a leader election object would be defined as the identity of the currently elected leader. The identity is not stored at any particular location; rather, it materializes as a stream of messages of the form *elected(x)* concurrently produced by the proxies involved in executing this protocol, and concurrently consumed by instances of the application using this protocol, on different machines distributed across the network.

- **Interfaces** (**endpoints**). The *interface* of a live distributed object is defined by the types of interfaces exposed by its proxies; these may include event channels and various types of graphical user interfaces. Interfaces exposed by the proxies are referred to as the live distributed object's *endpoints*. The term *endpoint instance* refers to a single specific event channel or user interface exposed by a single specific proxy. To say that a live object *exposes* a certain endpoint means that each of its proxies exposes an instance of

this endpoint to its local environment, and each of the endpoint instances carries events of the same types (or binds to the same type of a graphical display).

- **References**. The *reference* to a live object is a complete set of serialized, portable instructions for constructing its proxy. To *dereference* a reference means to locally parse and follow these instructions on a particular computer, to produce a running proxy of the live object. Defined this way, a live object reference plays the same role as a Java reference or a C/C++ pointer; it contains complete information sufficient to *locate* the given object and interact with it. Since live distributed objects may not reside in any particular place (but rather span across a dynamically changing set of computers), the information contained in a live distributed object's reference cannot be limited to just an address. If the object is identified by some sort of a globally unique identifier (as might be the case for publish-subscribe topics or multicast groups), the reference must specify how this identifier is resolved, by recursively embedding a reference to the appropriate name resolution object.

- **Types**. The *type* of a live distributed object determines the patterns of external interactions with the object; it is determined by the types of endpoints and graphical user interfaces exposed by the object's proxies, and the patterns of events that may occur at the endpoints. The constraints that the object's type places on event patterns may span across the network. For example, type *atomic multicast* might specify that if an event of the form *deliver(x)* is generated by one proxy, a similar event must be eventually generated by all *non-faulty* proxies (proxies that run on computers that never crash, and that never cease to execute or are excluded from the protocol; the precise definition might vary). Much as it is the case for types in Java-like languages, there might exist many very different implementations of the same type. Thus, for example, behavior characteristic to *atomic multicast* might be exhibited by instances of distributed protocols such as virtual synchrony or Paxos.

The semantics and behavior of live distributed objects can be characterized in terms of distributed data flows [10]; the set of messages or events that appear on the instances of a live object's endpoint forms a distributed data flow.

3.1 Quicksilver Scalable Multicast and Ricochet Real Time Multicast

Two important pre-built live objects are those that encapsulate Cornell's Quicksilver Scalable Multicast and Ricochet Real Time Multicast protocols. These two protocols were created in prior work, and then converted into live objects during our effort. They are key workhorse components of the typical application, carrying traffic on behalf of the objects that render graphics or capture information like images and GPS coordinates.

In this section we will say a bit about the problems each of these multicast objects solves and how they do so. Before doing this, however, it may be useful to briefly look at the architecture of the live objects platform, so that the reader can orient him or herself about the functionality of the various layers of the system per-se.

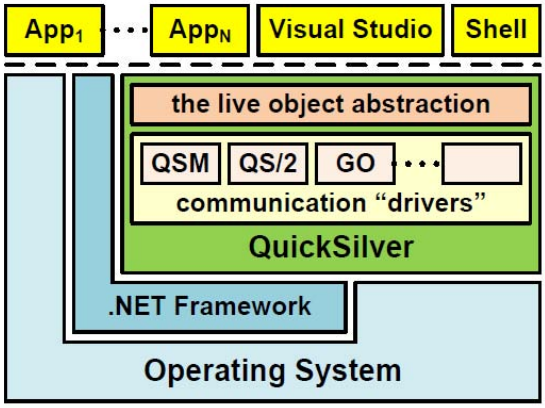

Figure 3 Layers of the Live Objects platform.

The resulting system is built in layers (Figure 3 which assumes that Quicksilver Scalable Multicast [18] is being used for replicated updates). The user's code and applications run as normal Windows applications in the .NET framework, which itself is implemented over the Windows Operating System (yellow boxes on top, blue below). Quicksilver Scalable Multicast is a multicast protocol that runs on the Windows system; one of several that can play this role.

The live objects abstraction is a set of extensions to the Windows .NET infrastructure that mediates when objects are launched, when they are composed, and when runtime type-checking actions are required. The basic focus of this layer is on the embedding of our new object interfaces into the.NET Common Language Runtime, and the hooks connecting the objects to the .NET type system and to the Windows shell (the GUI that interprets mouse actions, such as right-clicking on a file).

As can be seen in the illustration, Quicksilver Scalable Multicast is a free-standing multicast protocol that runs on Windows, but it has been "wrapped" with APIs (Application Programming Interfaces) that allow us to use it as a Live Object. One can also bring other protocols into the solution simply by using the same standard API, and we have done so for the Ricochet Real Time Multicast protocols [20] (not shown in figure 3). Other possible protocols that might be created in the future could include a multicast protocol focused on wireless mobility within tactical networks; such a step would make it possible to "port" a live objects application created in a wired network so that it would run properly in a tactical environment.

Briefly, here's a summary of the abstract functionality provided by a live object:

1. Each object has an Internet-wide unique name and is registered in a global distributed "directory" resembling a DNS (Domain Network Service), where it can be located by the software components that wish to access it.
2. Live objects have "distributed" types. Among other things, type-checking helps verify that the communication "type" beneath an object matches the developer's intent.

3. A live object provides a "natural" interface to its clients. An object representing a video stream would provide methods to send or receive video frames. An object that represents a replicated variable would provide get/set methods, etc.
4. Each object also has object-specific "logic" implementing some abstraction: a room, a medical record, a gossip-based overlay network, etc. This logic is translated into code that is deployed on and executed by the components accessing the live object. The protocols for replicating the object's data among components and for multicasting events among them are provided by type-specific underlying "communication drivers".
5. When a component tries to access a live object, an authentication and join protocol executes. This is used to enforce security, fetch the object's "logic" from the distributed live objects directory, obtain a snapshot of the object's state, initialize it etc. The object's code maintains its state in response to event notifications.

The underlying communications layer is modeled as a set of "pluggable" communication substrates, which play a role analogous to that of a device driver. Each live object needs a "communication driver" to replicate its state, propagate events and updates; different objects might use different drivers specialized for different settings [wireless, WAN (Wide Area Network), LAN (Local Area Network)] or properties (secure, low latency, etc). QuickSilver allows existing multicast and group communication toolkits to be adapted for use as communication drivers. It also comes with a suite of built-in scalable and high-performance communication drivers for common types of applications, and designed to complement each other:

- **QSM** (Quicksilver Scalable Multicast) is optimized to support large numbers of live objects that may represent streams of events, such as video channels, file backup folders, or stock price update notifications in a trading system. The current version of QSM is designed for enterprise LANs and datacenters. It can support tens of thousands of event streams, hundreds of users, and transmissions at network speeds. Events are delivered reliably, and the system is robust under stress.

- **Ricochet** (Ricochet Real Time Multicast) is a protocol that offers slightly weaker reliability properties in exchange for extremely powerful real-time guarantees at very high data rates. The basic approach is to send each message as an IP multicast, but to use additional IP multicasts that carry forward error correction (FEC) codes and can recover lost data before the loss is even noticed. However, Ricochet doesn't provide the very strong "TCP-style" reliability of QSM, nor is it designed to scale as well.

- **GO** (Gossip Objects [6][8]) is an extension of the live objects system that was developed jointly by Cornell and a team at IRISA (Institute de Recherche en Informatique et Systèmes Aléatoires)and the University of Rennes, in France. It uses the so-called gossip

protocols to build live objects that can help systems manage themselves, set their own configuration parameters, and even diagnose and self-repair when failures occur. In this manner, a wide range of gossip protocols can be integrated into the live objects setting.

We noted earlier that there are also live objects that interconnect via GIG and Web Services protocols to standard data sources such as Google Maps, Yahoo Weather, Microsoft MSN Streets, the FAA and other web sites. We expect this list to grow over time, with protocols to support wireless communication, IPTV (Internet Protocol Television), security protocols, and protocols with real-time properties. Obviously, for these kinds of protocols, events occurring on the client (such as movement that changes a user's location) must be relayed into the data center and then echoed back out to other replicas, hence these are layers that will freeze up if the back link to the data center becomes unavailable.

3.2 Extreme Scalability

Live objects could become popular, and one lesson of the modern Internet is that popular technologies must scale in a flexible, elastic manner. We tackled this question in the context of live objects that run over the Quicksilver Scalable Multicast layer (QSM). Although brevity precludes a highly technical discussion of scalability, a great deal of research went into the problem, and we summarize the approach taken and results obtained (see [18] for details).

Recall that QSM's role is to reliably multicast data in support of higher level "events" related to a number of live objects, such as updates to their state, commands that cause actions to occur, etc. In a typical datacenter scenario, some computers might be using large numbers of live objects, and different live objects could be shared by the same computers. The groups of computers on which these live objects "live" might thus overlap. The world of QSM is one of vast numbers of "overlapping" groups.

Let's make a simplifying assumption (we can remove it later). We'll assume that the overlap is clean and hierarchical. For example, there might be "big" objects A, B, and C directly superimposed, i.e. such that the groups of computers using them overlap perfectly. The hierarchy could also include subsets. For example, perhaps object D lives on half of these computers, and E on the other half. QSM starts by decomposing such a hierarchy of overlapping groups into a set of *regions*, by clustering computers based on their "interest". Each region contains computers that use the same live objects.

QSM now constructs a data dissemination overlay for each region. If possible, IP multicast is employed for this purpose (e.g. each region could be assigned its own IP multicast address). However, since IP multicast isn't always available, QSM can also run on some form of overlay

multicast technology. Dissemination is *unreliable*: like a UDP transmission in the Internet, a message might reach none of its destinations, some of them, or (if we are really lucky) all.

Next, QSM builds a peer-to-peer repair structure within each region. This involves many subtle issues, but the basic approach starts by constructing a logical token-ring that links the computers in the region (if a region gets larger than about 25 computers, we break the ring into a tree of smaller rings, linked by a higher-level ring). As the token travels around the ring, a few times per second, we efficiently encode the received message set of each computer. A machine that has a copy of a message some other machine lacks forwards a copy. If an entire region lacks a multicast, the sender re-multicasts it. In our experiments this is very rare; most packet loss involves a single machine that drops a single packet or a few in a row, and can be repaired locally with the help of a nearby peer.

Clustering computers into "regions" enables QSM to handle large numbers of live objects efficiently, by amortizing overhead. While in a traditional system, tens of thousands of live objects would involve tens of thousands of multicast protocols, QSM can use a smaller number of token rings, each of which works for multiple objects at once.

To handle computer crashes, QSM incorporates a hierarchical status-monitoring service structured a bit like the Internet DNS, but designed to track the health of components (live or failed), as well as some additional information used within our protocols. A consensus protocol is employed to ensure that the membership decisions will be consistently reported.

Obviously, we're skipping a lot of details, such as the flow-control mechanism used, data aggregation when multiple small messages are sent to the same group, etc (interested readers can find more information in the technical reports on our web site). But the upshot is that QSM seems to break every performance record we're aware of. We've scaled it to many thousands of groups (live objects) per computer, supported groups with hundreds members, and are able to saturate 100Mbit Ethernet interconnects with inexpensive PCs on the endpoints. The system is stable under stress, and very well tolerates load fluctuations, broadcast storms, or other degenerate behaviors. Moreover, even at the highest loads, overheads are quite low.

Now, all of this reflected a simplifying assumption, namely that groups are hierarchical. But it turns out multiple QSM hierarchies can be superimposed (Fig. 4). We're finding that even very irregular sets of overlapping groups can be "covered" with a surprisingly small number of hierarchies, provided that groups have Zipf-like popularity and traffic levels. For example, studies of financial instruments show that the i'th most popular stock or bond tends to be popular in proportion to $1/i^{\alpha}$ where the exponent, α, can be as large as 2.5 to 3.5. It seems reasonable to assume that in the Active Web, if applications use large numbers of live objects in irregular ways, the same property would hold.

In further work, we extended this basic idea to look at other cases in which correlation patterns of similar kinds can be leveraged to optimize the behavior of communication systems. As of the

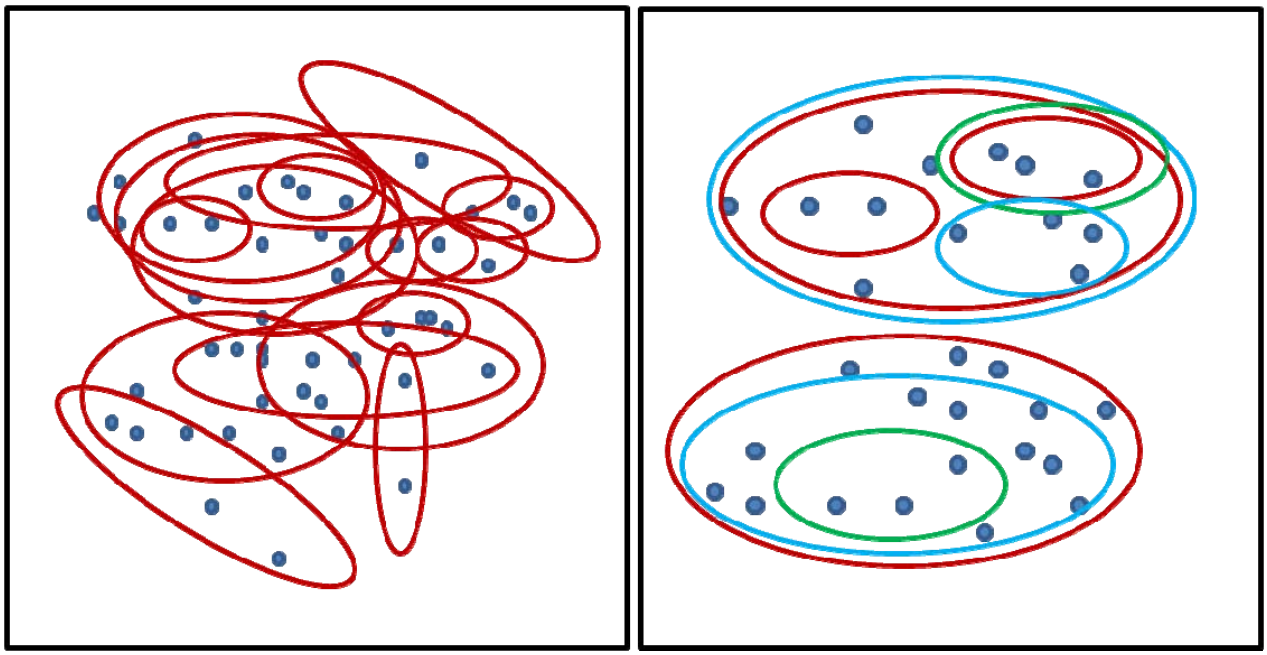

Figure 4 Key to QSM scalability is an algorithm for finding hierarchical structure (right) in overlapping groups (left).

end of this grant, we've used the approach to solve the following questions, in each case giving rigorous machine-learning solutions that "learn" by observing the actual patterns of correlation in a system, then applying an optimization that allocates some scarce resource in an optimal manner, and finally using the result to control the underlying system [7]:

- Managing IP multicast addresses in large-scale settings (Dr. Multicast)
- Merging similar object groups into larger groups to reduce the per-group overheads
- Performing flow control within multicast groups (AJIL)
- Selecting a communication target when a platform is asked to send messages periodically and has multiple "clients" issuing those requests, with different biasing functions. Here we make optimal target selections, subject to a utility function.
- Deciding how to group data into larger messages, so that instead of sending a single small message, we can send a larger message that might carry information on behalf of multiple applications, each with its own priorities and perhaps with different layouts of group members.
- Building information dissemination overlays whereby data centers or other sources can share links to send data in WAN settings, overcome obstacles such as bottlenecks or firewalls, and ensure reliability without sending redundant data that overloads members pointlessly.

3.3 Gossip Objects

A second approach to scaling was explored in a joint project we undertook with researchers in France, at IRISA/University of Rennes. Here, the idea was to use "gossip" multicast protocols to achieve scalability with very stable and predictable loads [6][8].

The GO (Gossip Objects) subsystem of the live objects distribution builds a random peer selection layer and then constructs a variety of gossip and epidemic mechanisms over it, presenting these as live objects that can perform tasks such as tracking overall system state, assisting in auto-configuration or repair, and constructing the overlay networks needed for QSM's dissemination layer. The key problem here involves optimization. First, because multiple objects run on the same machines, whereas normal gossip protocols gossip with peers selected on a per-application basis, GO must select a good peer for each round of gossip in a manner that optimizes over all applications. Secondly, because a single message can often carry updates from multiple objects, GO must run an optimization protocol to decide which objects to include in each message it sends.

3.4 What's in a type?

Live objects open the possibility of extending normal type systems to encompass distributed behavioral patterns [19]. A replicated variable that represents an account balance in a banking system might need strong reliability and fault-tolerance properties such as virtual synchrony, while for a similar variable in a monitoring application, weaker reliability properties and scalability of gossip might be a better match. Thus, to make live objects truly useful, we need a way to describe such behaviors as a part of their types.

In QuickSilver, the type of a live object is a tuple, the elements of which specify different "aspects" of the type (much like in aspect-oriented programming). One element is the object's interface (in the usual sense). Another is its "category" (replicated service, replicated variable, event stream, gossip object etc.), which tells which communication protocol driver to use. Other aspects configure the underlying dissemination substrate, and specify the object's reliability, fault-tolerance, and security properties.

By expressing distributed types this way, we enable a next step in which type information could be used as part of the application design and implementation process. A development environment such as Visual Studio would "understand" the possible distributed behaviors of such a typed object, and could guide the developer through the process of implementing code that will run correctly under the assumption that the communication drivers underlying the live

object implement the specified behaviors. Moreover, the runtime system can throw exceptions if mistakes are made, for example if a component designed to work correctly only with live virtually synchronous multicast streams tried to access a live multicast stream that has a weaker QSM or best-effort reliability property.

Type-based programming tools and debugging tools have transformed the experience of building applications for desktop environments. Service oriented architectures (which also revolve around type systems, albeit simple ones) are having a similar impact in networked applications that interact with services hosted in datacenters. Live objects are a natural step in this direction, and bring the benefits of strong typing to the realm of decentralized peer-to-peer applications. They suggest that the active web could be far more than just a veneer over the same old Internet technologies.

3.5 Quicksilver Properties Framework

Earlier, we commented that the Properties Framework [11] extends QSM by allowing some groups to have stronger reliability properties, defined using a high-level declarative *Properties Language*. Using this language, we are able to support a number of important reliability models – "reliability types". Let's look at two of these models and how they might assist the developer of the "village sweep" mission discussed earlier:

- **Virtual Synchrony** is a powerful distributed computing model in which active programs join *process groups*, within which multicasts are used to disseminate updates and other events. At the moment a process joins, it can initialize itself using a *state transfer* from some active member, and the virtual synchrony platform notifies all members each time membership changes. The power of this model is that it can support consistency guarantees: all the users of a virtual synchrony group see the same thing in the same order. This can be important when multiple users are concurrently taking actions, and yet the "physical world" needs to somehow order them. For example, suppose that several threats are present and we want different soldiers to respond to different threats. With virtual synchrony, the Quicksilver platform would enforce a single, system-wide event ordering, hence all participants see the same events in the same sequence. This can then drive a clean partitioning of roles.

- **Transactions** are a stronger execution model. Suppose that a power failure were to cause a few machines to crash simultaneously. Although we didn't make this clear in our discussion of virtual synchrony, that model only provides consistency among live objects that remain operational. Transactional live objects support the full one-copy serializability version of the ACID model; with this guarantee, crashed objects can reconstruct a consistent, agreed-upon state after they recover. However, these stronger guarantees

come at a (steep) price: performance and scalability won't be nearly as good. In our scenario, transactional semantics would be used when recording situational state that must be preserved for later analysis, and in which multiple soldiers might be updating the same information at the same time. Transactions guarantee consistency, tolerance of faults and durability, provided that enough replicas survive to permit recovery of the situation state records.

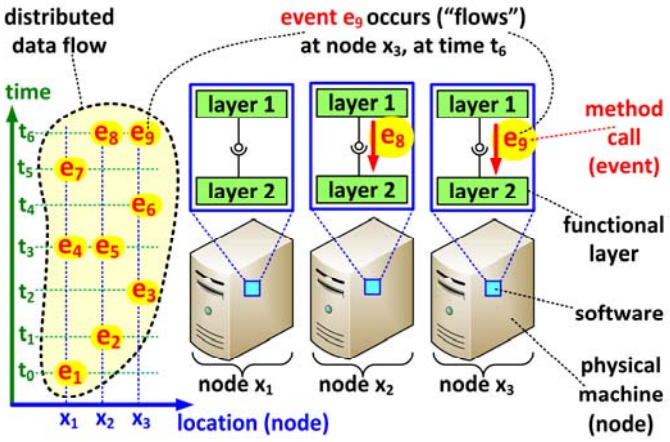

Figure 5 Distributed Event Flow in the Properties Framework

In the Properties Framework, each object can be configured to have the kind of consistency guarantee appropriate to the way it will be used. Different models are expressed using a kind of script that the framework "executes" to control the delivery of messages and other events, and to trigger actions such as the forwarding of a message to repair a loss, or the sending of ordering information when a batch of messages must be placed into a total order.

The term distributed data flow (also abbreviated as distributed flow) refers to a set of events (see Figure 5) in a distributed application or protocol that satisfies the following informal properties:

Asynchronous, non-blocking, and one-way. Each event represents a single instance of a non-blocking, one-way, asynchronous method invocation or other form of explicit or implicit message passing between two layers or software components. For example, each event might represent a single request to multicast a packet, issued by an application layer to an underlying multicast protocol. The requirement that events are one-way and asynchronous is important. Invocations of methods that may return results would normally be represented as two separate flows: one flow that represents the requests, and another flow that represents responses.

Homogeneous, unidirectional, and uniform. All events in the distributed flow serve the same functional and logical purpose, and are related to one-another; generally, we require that they represent method calls or message exchanges between instances of the same functional layers, or instances of the same components, but perhaps on different nodes within a computer network. Furthermore, all events must flow in the same direction (i.e., one type of a layer or component always produces, and the other always consumes the events), and carry the same type of a payload. For example, a set of events that includes all multicast requests issued by the

same application layer to the same multicast protocol is a distributed flow. On the other hand, a set of events that includes multicast requests made by different applications to different multicast protocols would not be considered a distributed flow, and neither would be a set of events that represent multicast requests as well as acknowledgments and error notifications.

Concurrent, continuous, and distributed. The flow usually includes all events that flow between the two layers of software, simultaneously at different locations, and over a finite or infinite period of time. Thus, in general, events in a distributed flow are distributed both in space (they occur at different nodes) and in time (they occur at different times). For example, the flow of multicast requests would include all such requests made by instances of the given application on different nodes; normally, such flow would include events that occur on all nodes participating in the given multicast protocol. A flow, in which all events occur at the same node would be considered degenerate.

Formally, we represent each event in a distributed flow as a quadruple of the form (x,t,k,v), where x is the location (e.g., the network address of a physical node) at which the event occurs, t is the time at which this happens, k is a version, or a sequence number identifying the particular event, and v is a value that represents the event payload (e.g., all the arguments passed in a method call). Each distributed flow is a (possibly infinite) set of such quadruples that satisfies the following three formal properties.

1. For any finite point in time t, there can be only finitely many events in the flow that occur at time t or earlier. This implies that in which flow, one can always point to the point in time at which the flow originated. The flow itself can be infinite; in such case, at any point in time, eventually a new event will appear in the flow.
2. For any pair of events e_1 and e_2 that occur at the same location, if e_1 occurs at an earlier time than e_2, then the version number in $e_1$must also be smaller than that of e_2.
3. For any pair of events e_1 and e_2 that occur at the same location, if the two events have the same version numbers, they must also have the same values.

In addition to the above, flows can have a number of additional properties.

- *Consistency.* A distributed flow is said to be consistent if events with the same version always have the same value, even if they occur at different locations. Consistent flows typically represent various sorts of global decisions made by the protocol or application.
- *Monotonicity.* A distributed flow is said to be weakly monotonic if for any pair of events e_1 and e_2 that occur at the same location, if v has a smaller version than e_2, then e_1 must carry a smaller value than e_2. A distributed flow is said to be strongly monotonic (or simply monotonic) if this is true even for pairs of events e_1 and e_2 that occur at different locations. Strongly monotonic flows are always consistent. They typically represent

various sorts of irreversible decisions. Weakly monotonic flows may or may not be consistent.

Distributed data flows serve a purpose analogous to variables or method parameters in programming languages such as Java, in that they can represent state that is stored or communicated by a layer of software. Unlike variables or parameters, which represent a unit of state that resides in a single location, distributed flows are dynamic and distributed: they simultaneously appear in multiple locations within the network at the same time. As such, distributed flows are a more natural way of modeling the semantics and inner workings of certain classes of distributed systems. In particular, the distributed data flow abstraction has been used as a convenient way of expressing the high-level logical relationships between parts of distributed protocols.

4.0 CONCLUSION - AN ARCHITECTURE FOR THE ACTIVE WEB

The Cornell AFRL funding permitted a complete prototype of the live objects system to be created, tested extensively in the lab at Cornell, and shared through FreeBSD licensing in source form for use on Windows and Linux platforms. Several real applications were created, including a number that were designed jointly with our colleagues at AFRL and at the office of the CIO/CTO of the Air Force in Washington, and demonstrate how live objects would function in real military deployments. There have been substantial numbers of downloads by interested developers. As noted earlier, this communications driver is breaking every performance record we're aware of in the technology space: raw throughput, scalability, tolerance of stress, and robustness against broadcast storms and other forms of oscillatory behavior. The remainder of the system (including the properties framework) is running as an experimental prototype and tested under laboratory conditions. Other elements include a configuration manager service that can run at Internet Scale, the GO subsystem mentioned above, and many elements of the runtime and debugging/performance-tuning environment.

We believe that live objects enable a completely new kind of distributed programming, inspired by the Web, but in which much of the content is dynamic and may be evolving in real-time[12][9]. Live objects could represent video feeds, streams of media or other content generated by participating computers, telemetry from sensors, etc. By integrating such content into systems like Windows or J2EE (Java 2 Enterprise Environment) in a clean and natural way that leverages the power of type systems and component integration technologies, but offers a portal to distributed computing, we're hoping to enable a revolution. Live objects could open the door to a new and disruptive generation of Active Web applications that combine high data rates with strong properties, including fault-tolerance, consistency, and security.

5.0 PRESENTATIONS

Our research team is often invited to present work through lectures in various venues. The live objects work was presented in talks at more than a dozen major universities, keynote talks at several conferences and at many smaller workshops. In addition, our published papers were presented in sessions at the corresponding conferences.

Dialog with Air Force Staff

Our work came to the attention of the Air Force CIO/CTO office in late 2008, when that group was asked to study options for supporting disconnected information-enabled missions with full dynamic situational awareness and encountered the GIG reach-back limitation mentioned earlier. The CIO and CTO asked us to work with their DC-based teams to understand how live objects might be used to educate the Air Force vendor community about the power of these new technical options, how they work and how to incorporate them into military products that must support GIG standards.

In this connection, a series of meetings during 2009 and 2010 were held in the offices of the CIO at the Pentagon and those of the CTO in Arlington. The Cornell team created a number of specialized demonstrations and ran them at the Pentagon and in the CTO's office, using multiple laptops and our own networking equipment so as to comply with military network isolation rules. Presentations were also incorporated into the Cornell/AFRL workshop series, in which Cornell researchers visited AFRL Information Directorate at Rome, New York on a regular schedule, teaching mini-courses that helped AFRL research staff understand the technologies involved, the underlying science, and to master the actual use of the live objects platform per-se. Cornell also assisted AFRL in developing materials to present the findings of the effort to the SAB (Scientific Advisory Board) during their 2009 review of the AFRL effort.

6.0 RESEARCHERS SUPPORTED

Kenneth Birman, Professor
Robbert Van Renesse, Principal Research Scientist
Daniel Freedman, Postdoctoral Associate
Krzysztof Ostrowski, Postdoctoral Associate, PhD Degree
Ymir Vigfusson, Graduate Research Assistant, 10/15/09, PhD Degree, Postdoctoral Associate at IBM Research, Haifa
Robert Burgess, Graduate Research Assistant, 8/15/09
Haoyuan Li, Graduate Research Assistant, 8/15/09, MS Degree, Software Engineer at Conviva
Lonnie Princehouse, Graduate Research Assistant, 8/15/09

Daniel Williams, Graduate Research Assistant, 8/15/09
Hussam Abu-Libdeh, Graduate Research Assistant, 5/15/09
Tudor Marian, Graduate Research Assistant, 12/31/08, PhD Degree, Postdoctoral Associate, Cornell
Qi Huang, Postdoctoral Associate, PhD student starting Spring 2011
Tudor Marian, Graduate Research Assistant, 12/31/08, PhD Degree

7.0 SOFTWARE RELEASE

Our platform can be downloaded from http://liveobjects.codeplex.com/ .

APPENDIX A - PUBLICATIONS

Our work yielded a number of publications, all of which have been uploaded to the Jiffy site and are available to the public with no restrictions or limitations. The following table lists the ones specifically tied to the live objects effort.

2010

[1] Kevlar: A Flexible Infrastructure for Wide-area Collaborative Applications. Qi Huang, Daniel A. Freedman, Ymir Vigfusson, Ken Birman, and Bo Peng. ACM/IFIP/Usenix 11th International Middleware Conference (Middleware 2010), Bangalore, India, Nov 29 - Dec 3, 2010.

[2] Enabling Tactical Edge Mashups with Live Objects. Daniel Freedman, Ken Birman, Krzys Ostrowski, Mark Linderman, Robert Hillman, Albert Frantz Proceedings of the 15th International Command and Control Research and Technology Symposium (ICCRTS '10), Information Sharing and Collaboration Processes and Behaviors Track. Santa Monica, CA, USA. June 22-24, 2010. Best paper in track; Best paper in conference.

[3] Dr. Multicast: Rx for Data Center Communication Scalability. Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Robert Burgess, Haoyuan Li, Gregory Chockler, Yoav Tock. Eurosys, April 2010 (Paris, France). ACM SIGOPS, pp. 349-362.

[4] Quilt: A Patchwork of Multicast Regions. Qi Huang, Ken Birman, Ymir Vigfusson, Haoyuan Li. 4th ACM International Conference on Distributed Event0Based Systems (DEBS2010). Cambridge, United Kingdom. July 2010.

2009

[5] Self-Replicating Objects for Multicore Platform. Krzysztof Ostrowski, Chuck Sakoda, and Ken Birman. 24th European Conference on Object-Oriented Programming (ECOOP 2010).LNCS Volume 6183/2010, 452-477.

[6] GO: Platform Support for Gossip Applications. Ymir Vigfusson, Qi Huang, Ken Birman, Deepak Nataraj, ACM TOCS. October 2009.

[7] Affinity in Distributed Systems. Ymir Vigfusson, PhD dissertaton. Cornell University, Sept. 2009. (Degree conferred Feb. 2010)

[8] GO: Platform Support For Gossip Applications. Ymir Vigfusson, Ken Birman, Qi Huang, Deepak P. Nataraj. IEEE P2P 2009. Seattle, WA. September 9 - 11. pp: 222-231.

[9] Storing and Accessing Live Mashup Content in the Cloud. Krzysztof Ostrowski and Ken Birman. 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS 2009). Volume 44, Issue 2, April 2010.

[10] Programming Live Distributed Objects with Distributed Data Flows. Krzysztof Ostrowski, Ken Birman, and Danny Dolev. Technical Report, March 2009.
[11] Implementing Reliable Event Streams in Large Systems via Distributed Data Flows and Recursive Delegation. Krzysztof Ostrowski, Ken Birman, DannyDolev, and Chuck Sakoda. 3rd ACM International Conference on Distributed Event-Based Systems (DEBS 2009). Nashville, TN, USA. July 6-9, 2009.
[12] WS-OBJECTS: Extending Service-Oriented Architecture with Hierarchical Composition of Client-Side Asynchronous Event-Processing Logic. Krzysztof Ostrowski and Ken Birman. 7th IEEE International Conference on Web Services (ICWS 2009). Los Angeles, CA, USA. July 9-10, 2009. pp: 25-34.
[13] Live Distributed Objects for Service Oriented Collaboration. Ken Birman, Jared Cantwell, Daniel Freedman, Qi Huang, Petko Nikolov, Krzysztof Ostrowski. Third International Conference on Intelligent Technologies for Interactive Entertainment (Intetain '09), Demo Track. Amsterdam, The Netherlands. June 22, 2009.
[14] Edge Mashups for Service-Oriented Collaboration. Ken Birman, Jared Cantwell, Daniel Freedman, Qi Huang, Petko Nikolov, and Krzysztof Ostrowski. IEEE Computer. Volume 42, Number 5, pgs 92-96. May 2009
[15] Building Collaboration Applications That Mix Web Services Hosted Content with P2P Protocols. Ken Birman, Jared Cantwell, Daniel Freedman, Qi Huang, Petko Nikolov, Krzysztof Ostrowski. In the proceedings of IEEE International Conference on Web Services (ICWS). Los Angeles, CA. July 6-10, 2009.

2008

[16] SOLO: Self Organizing Live Objects. Qi Huang (Huazhong University of Science and Technology), Ken Birman. Technical Report. December 2008.
[17] Using Live Distributed Objects for Office Automation. Jong Hoon Ahnn, Ken Birman, Krzysztof Ostrowski, Robbert van Renesse. In proceedings of the ACM/IFIP/USENIX 9th International Middleware Conference. Leuven, Belgium. December 2008.

Proof of concept work completed prior to the effort and cited in the text

[18] QuickSilver Scalable Multicast (QSM). Krzysztof Ostrowski, Ken Birman, Danny Dolev. 7th IEEE International Symposium on Network Computing and Applications (IEEE NCA 2008). Cambridge, MA. July 2008.
[19] Programming with Live Distributed Objects. Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. In Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008). Cyprus. July 2008. J. Vitek, Ed. Lecture Notes In Computer Science, vol. 5142. Springer-Verlag, Berlin, Heidelberg, 463-489.
[20] Ricochet: Lateral Error Correction for Time-Critical Multicast. Mahesh Balakrishnan, Ken Birman, Amar Phanishayee, and Stefan Pleisch. In Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07). Cambridge, MA. April 2007.

LIST OF ABBREVIATIONS, AND ACRONYMS

AFRL	Air Force Research Laboratory
API	Application Programming Interface
C2	Command and Control
CAC	Computer Access Card
CIO	Chief Information Officer (of the US Air Force)
CTO	Chief Technology Officer (of the US Air Force)
DNS	Domain Network Service
ESB	Enterprise Service Bus
FAA	Federal Aviation Administration
FEC	Forward Error Correction
FIFO	First In First Out
FreeBSD	Free Berkeley Standard Distribution (public domain licensing)
GIG	Global Information Grid
GO	Gossip Objects
GPS	Global Positioning System
GUI	Graphical User Interface
IC2	Integrated Command and Control
IP/IPv4	Internet Protocol (version 4.0)
IP multicast	Internet multicast protocol
IPTV	Internet Protocol Television
IRISA	Institute de Recherche en Informatique et Systèmes Aléatoires
J2EE	Java 2 Enterprise Environment
JFEX	Joint Forces Expeditionary Exercise
LAN	Local Area Network
.NET	The Windows managed execution environment
QSM	Quicksilver Scalable Multicast
SAB	Scientific Advisory Board
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Universal Resource Locator
WAN	Wide Area Network
WS	Web Services
XML	Extensible Markup Language
XNA	The Windows 3D graphical rendering subsystem